Final Report
DrinkIT

Kajsa Bjäräng, Viktoria Enderstein, Elin Eriksson, Lisa Fahlbeck, Alice Olsson

2018-10-28
Version 7

Table of Contents

1 Introduction

People are always searching for the next great game to play during parties and predrinks. There are many subpar options available, and the goal is to combine and enhance many of these existing games to make the ultimate party-game.

The game will be customizable when it comes to both players and content. Users will be able to enter the number of players and their names. Users can then choose which categories they want to play with for this round and set the length of the game. The categories are made up of a combination of several different games, all popular in different party settings, and a couple unusual ones to give the game depth and keep it exciting.

During the game players will be randomly chosen to perform different challenges, with the options to either "pass" or "fail", decided by the group. The challenges will be randomized from the chosen categories and will have differing levels of difficulty, reflected in the amount of points they give if the players succeed.

At the end of the game a scoreboard will be shown with the points all players have collected, along with the option to play again, which restarts the game, giving the players the choice to either play immediately again or change chosen settings beforehand.

1.1 Definitions, acronyms, and abbreviations

DrinkIT - The name of the application as well as the name of the main class in the model package.
Challenge - A task to perform for each player when it is their turn, for example answer a question or perform a song or charade.
Category - All challenges are divided into one of nine different categories.
GameRound - GameRound contains all information needed to distinguish and save statistics about any round during the game. It contains a player, a challenge, whether the challenge succeeded or failed, and a static list of GameRounds of the played rounds.
Gradle - An open-source build automation tool designed for multi-project builds. Gradle comes included in Android Studio.
Travis - Automatically builds the code and checks if everything still works after a push to GitHub. Otherwise it reports that something went wrong, which is good when several people work on the same project.
Android Studio - An integrated development environment (IDE). It's the official IDE for Google's Android Operating system.

JSON (JavaScript Object Notation) - A lightweight data-interchange format, which is both easy to write and to parse and generate. Based on a subset of JavaScript, but completely language independent.
Activity - In Android Studio, an Activity is a class with a connected xml file handling views.

2 Requirements

2.1 User Stories

In order to get a working game these five most important user stories were implemented. An overview of all user stories can be found in Appendix A.

2.1.1 User Story One

As a user, I want to add a player, so I can have multiple players.

Acceptance Criteria:

- ❖ The player exists in the list of players
- ❖ It is not possible to add the same player twice

Tasks:

- ❖ There is a form to enter a name
- ❖ There is a button to add the player

2.1.2 User Story Two

As a user, I want to be able to choose categories, so I can customise the game with challenges I like.

Acceptance Criteria:

- ❖ It is possible to see all different categories
- ❖ It is possible to only choose one category
- ❖ It is possible to choose multiple categories
- ❖ It is possible to choose any number of categories
- ❖ Only challenges from the chosen categories will be shown during the game.
- ❖ It's possible to remove a chosen category

Tasks:

- ❖ There exists a number of categories
- ❖ If a category is chosen there is a visual feedback
- ❖ There are different buttons to add different categories to the game
- ❖ There is a button to chose all categories immediately

2.1.3 User Story Three

As a user I would like to choose an approximate duration of the game, so I can decide for how long I would like to play.

Acceptance Criteria:

- ❖ When a game is started an option to choose the duration of the game is displayed.
- ❖ There are three options to choose from
- ❖ Three of the options are set amounts of rounds (short, middle, long game)
- ❖ Depending of the number of players and the chosen option for duration an even number of rounds is calculated.
- ❖ The calculated number of rounds, depending of the number of players and the chosen option is the exact number of rounds played before the game is finished.

Tasks:

- ❖ There are 3 buttons to choose the duration of the game
- ❖ The number of rounds is set according to the chosen duration button and the amount of players.

2.1.4 User Story Four

As a user, I want to see a challenge when I play so I can do the challenge.

Acceptance Criteria:

- ❖ I can see a card on screen each round
- ❖ A player is chosen to play each specific challenge
- ❖ The difficulty of the challenge is shown on screen
- ❖ The challenge shown belongs to one of the chosen categories
- ❖ The challenge shown has not been shown before during the active game

Tasks:

- ❖ Randomize a player
- ❖ Show the name of the player on the display.
- ❖ Randomize a task.
- ❖ Show the task on the display

2.1.5 User Story Five

As a user, I want different challenges to appear each round, so I can be sure I don't have to do the same challenge again during the same game.

Acceptance Criteria:

- ❖ A challenge shows up for each round
- ❖ All challenges that are from the chosen categories are available to appear in the beginning of the game

- ❖ Only challenges that are from the chosen categories and that have not been played are available to appear in the end of the game
- ❖ As long as there are unplayed challenges available in a chosen category, the players only see new challenges instead of the same challenge twice
- ❖ A player will never play the same challenge twice

Tasks:

- ❖ When a challenge has been played, it is removed from possible challenges to appear in coming rounds, until the list of challenges in a category run out
- ❖ Statistics are stored showing which player has played which rounds, so make sure that even if a challenge is shown twice, it doesn't happen to the same player.

2.2 User Interface

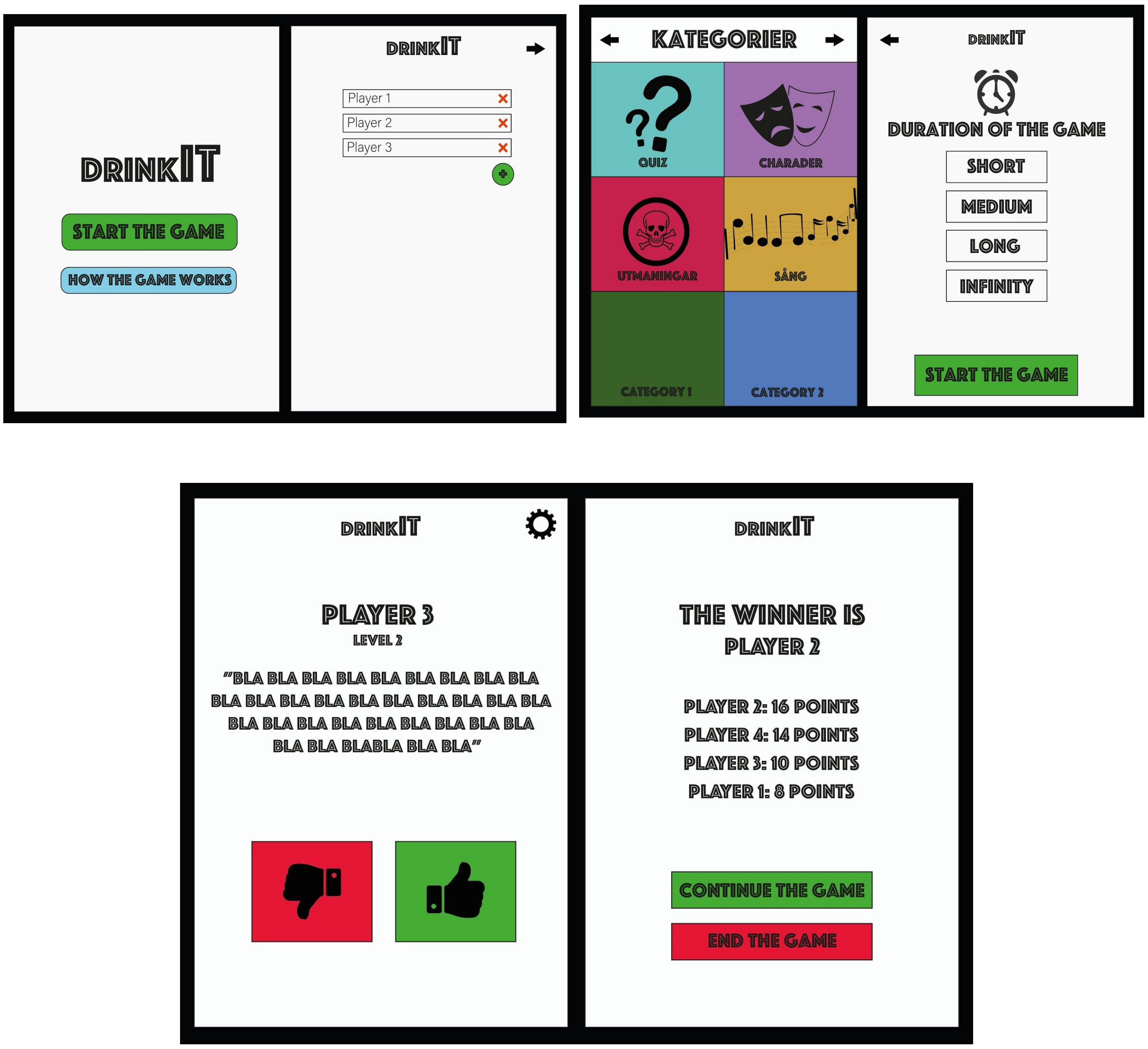
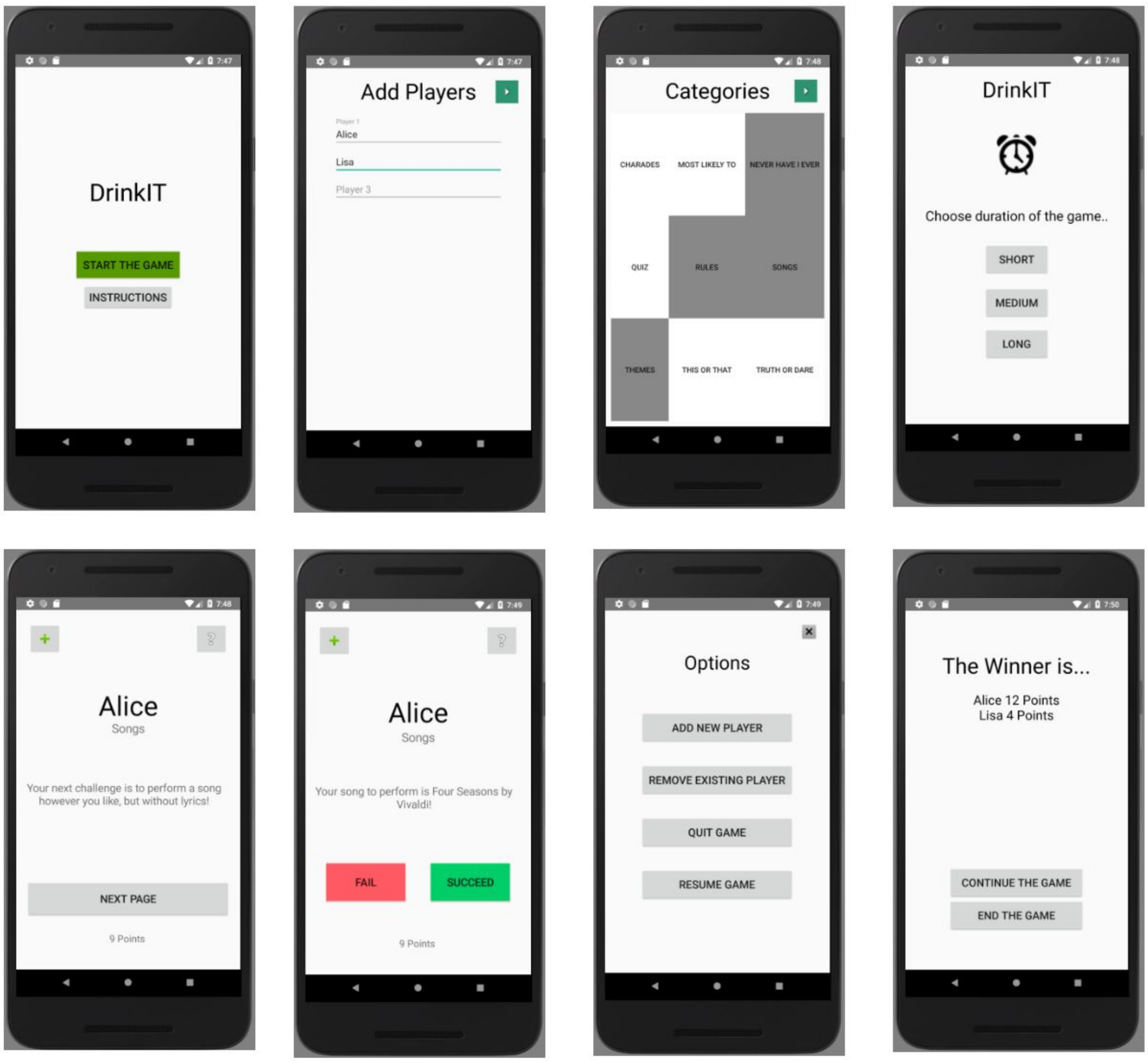


Figure 1. The first sketches of the GUI



Your next challenge is to perform a song however you like, but without lyrics!

Your song to perform is Four Seasons by Vivaldi!

Figure 2. Implemented graphical interface

3 Domain Model

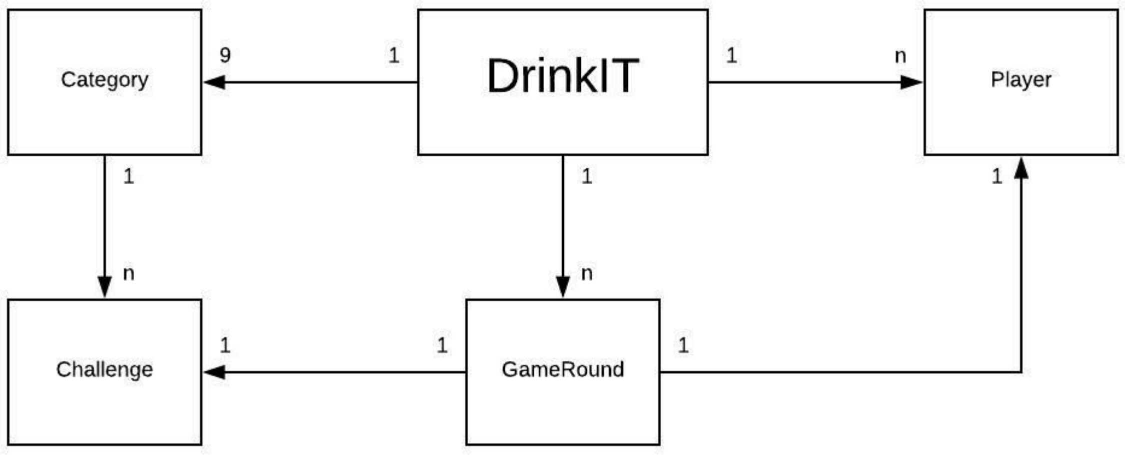


Figure 3. The domain model of the application.

3.1 Class responsibilities

The application consists of a single component. The model package of the application is represented in the domain model. DrinkIT is the main class in the model and handles most of the functionality for this package and also the communication from the model package and out to the controller.

DrinkIT contains of a list of players which each player has a name and a point. The class also contains a list of 9 categories which can be set to active and inactive depending on the players choice. Each category contains a name, instruction, the state of the category and a list of challenges.

The class GameRound connects the challenges and the players to make a game round. GameRound randomly choose one player and one challenge from an active category and present that to the player by DrinkIT that is connect to the controller which in turn is connected to the view and the information reaches the player. GameRound also has a static list of gamerouns which keeps the playedrounds to be able to keep statistics and for the game to control that one player doesn’t get the same challenge twice.

4 System architecture

The application is made in Android Studio version 3.1.4, written in the language Java. It uses the built in automation tool Gradle version 4.4 which comes included in this version of Android Studio. The application is designed to be played on a single unit of an android smartphone. DrinkIT targets API 27 and Android 8.1, and supports all newer versions of OS.

4.1 Game flow

To start the game the user has to add all participating players by name, then choose which categories should be included, and then decide the duration of the game, which is calculated to a specific number of rounds. Thereafter a challenge from one of the chosen categories is randomly shown on the screen. Some challenges give the player points if they are accomplished whilst others are just for fun. After the number of challenges corresponding to the chosen duration have been played the game is finished and a scoreboard is shown with an option to continue playing. During the game there are options always available to add another player, remove an existing player, quit the game, or access instructions and help.

5 System design

The application is designed to make it easy for future developers to understand, navigate and expand the code. To achieve this the development of the application is built on a number of design principles and patterns.

5.1.1 Single responsibility principle

Classes doing one thing only, and doing it well, leads to a robust code. In the application DrinkIT one of the goals was to make sure all of the classes and class methods follows the Single Responsibility Principle. For example most classes in the model follow the Single Responsibility Principle, however there are still exceptions at this point in time, and this is a future improvement to work on. DrinkIT is an example of a class that does not follow the Single Responsibility Principle, and the refactoring of removing functionality from this class has been started but not completed. The remaining classes in the Model follow the principle, as they only contain functionality regarding themselves.

5.1.2 Open Closed Principle

Making the application open for extension but closed for modification is achieved by the implementation of design patterns. The Model-View-Controller pattern makes it easy for future developers to expand DrinkIT, since the Controller class handles all communication between different packages and therefore makes it easy to add more components or packages. The implementation of JSON-files also makes it easy to add more challenges by only having to add them once.

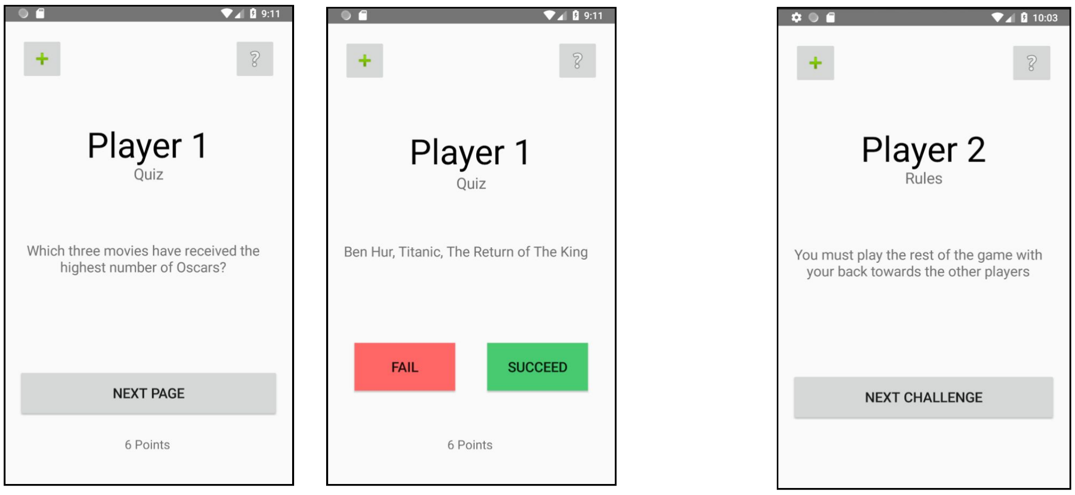
To make the application closed for modification, encapsulation has been used. The goal of encapsulation is to avoid global access of variables and methods which prevents data to be modified by an outside user. All variables and methods that do not need to be accessed outside of their classes are private, so they cannot be accessed and modified elsewhere in the code. This is an example of how the Open-Closed Principle is achieved in the application.

5.1.3 Model-View-Controller Pattern (MVC)

The goal of the dependency between classes is to achieve as high cohesion and as low coupling as possible. The MVC pattern helps keeping the coupling low by handling the communication between packages. Due to DrinkITs many views and game logic the implementation of the MVC pattern was a suitable choice.

The model of the application lies in the package Model where the class DrinkIT is the heart of the model. The View contains multiple Activities which lie in the View package and the Controller class lies in the Controller package and acts as a connection between the Model, View and Services packages.

The class ChallengePageOne gets information from the model regarding which is the current category, and then creates the view dynamically depending on that category. For example the category Quiz contains a question and an answer, while the category Rules only contains a challenge.


Player 1
Quiz
Which three movies have received the highest number of Oscars?
NEXT PAGE
6 Points

Player 1
Quiz
Ben Hur, Titanic, The Return of The King
FAIL
SUCCEED
6 Points

Player 2
Rules
You must play the rest of the game with your back towards the other players
NEXT CHALLENGE

Figure 4. View of a challenge from the category “Quiz” with two pages vs a view of a challenge from the category “Rules”.

The model is responsible for the domain logic and holds several classes with different responsibilities. The model consists of five classes, Category, Challenge, DrinkIT, GameRound and Player. They all hold different functionality, but collaborate as a unit with few dependencies.

The class DrinkIT acts as an interface to the Model package from the Controller. It’s the Controller which interacts with the players, through input from the different views, and calls for the right methods from the model.

5.2 Design model

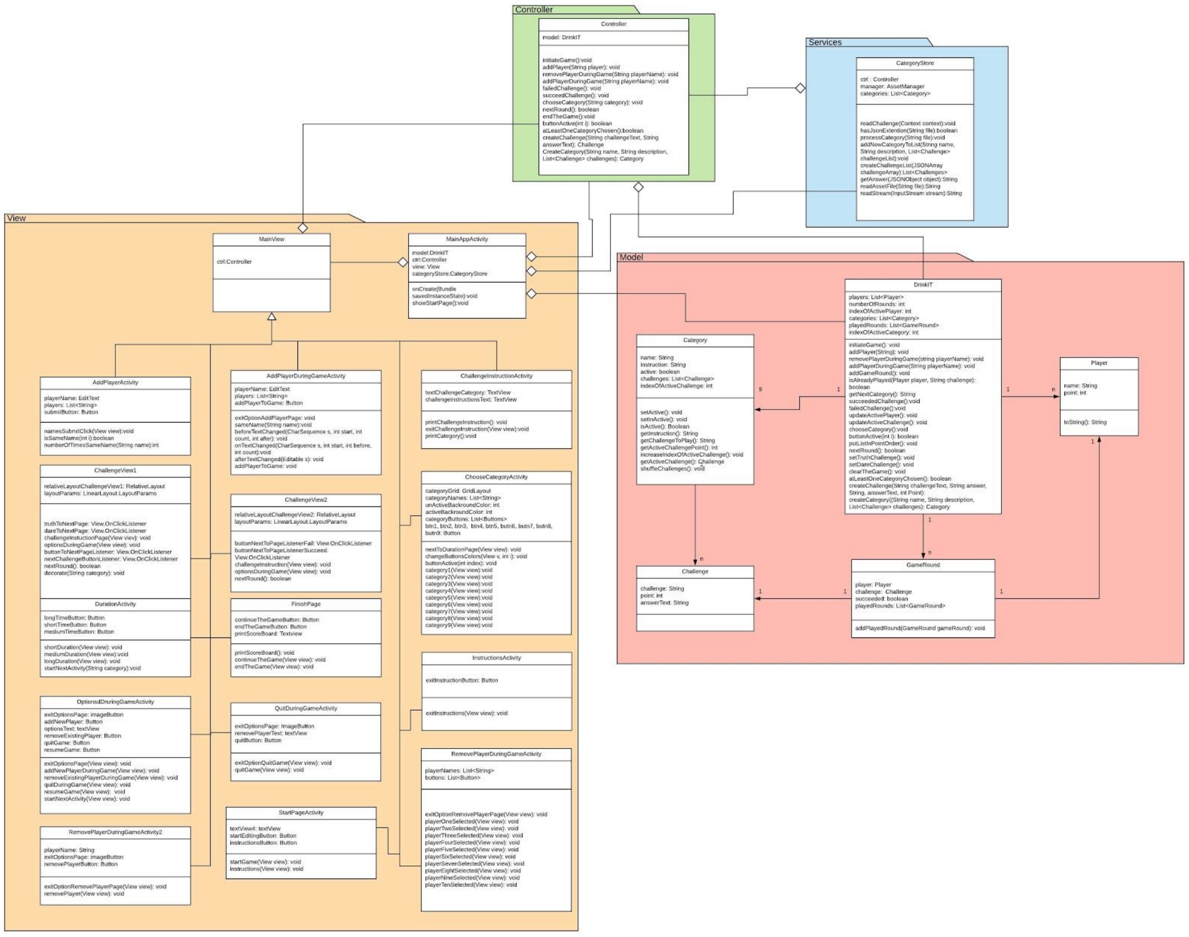


Figure 5. An overview of the whole project with all packages. For a more detailed view, see Appendix B, C and D.

The design model consists of four different packages, Services, View, Model and Controller and follows, as earlier mentioned, the design pattern MVC. The class DrinkIT which lies in the Model package holds a key position. You could say that the DrinkIT class is at the heart of the game since it holds most of the underlying game logic.

MainAppActivity initializes the game and creates an instance of each main class - Controller, MainView and the model DrinkIT - which are used throughout the game. MainView is created to decrease dependencies by acting as a superclass to all other views, thereby holding the only instance of the Controller available to all Views.

When starting the game the startpage is a simple view with two buttons acting as a clear entry point to the game. When starting the game the user gets to add all players by manually entering their names. This view is designed so that the user dynamically receives error messages with the help of the Observer Pattern, found in AddPlayerActivity, if the input entered is incorrect. The game is configured to accept a minimum of two players and no duplicate names.

DrinkIT contains a list of players. The class Player contains a name and a point, acting as a score. The players are added in the list in the beginning of the game as explained above. This list contains the players during the game and is used in different methods in DrinkIT.

DrinkIT also contains a list of categories. The class Category contains a name, an instruction on how to play the Category in question, a list of challenges, an index of the active challenge and a state, either active or inactive.

The user gets to choose from nine categories which they can include in the game. When selecting a category, its state gets set to active. DrinkIT contains an integer indexOfActiveCategory which is incremented after each game round to supply the user with the next challenge. Further on in the text, under *4. Persistent data management*, there can be found an explanation of how challenges are created.

GameRound is the class that assists the application in keeping track of all statistics regarding earlier played rounds. After every played round the player, the challenge and the outcome - if the played succeeded or failed - is stored in a list of game rounds i the GameRound class. DrinkIT uses the list to make sure the same player never has to play the same challenge twice.

To loop through the players and ensure each player gets to play the same number of rounds during the game, DrinkIT contains an integer, indexOfActivePlayer, which is incremented after each completed round. Every time the index reaches the last player in the list, the list is shuffled and the index is set back to 0. This is to avoid players consistently showing up in the same order, which makes the game more interesting and unexpected to play. There is also logic to prevent the same player showing up twice in a row, even as the list is shuffled and the index reset.

5.3 The test package

During the implementation of the application, tests have been implemented after every new implemented user story, to make sure the code that has been written is solid and works correctly. The application contains two test packages, com.TDA367.drinkit (test) and com.TDA367.drinkit (androidTest), which contains multiple unit tests, made with the framework JUnit. There is a 100% coverage of test on the classes but not on the methods when we didn't test get or set methods.

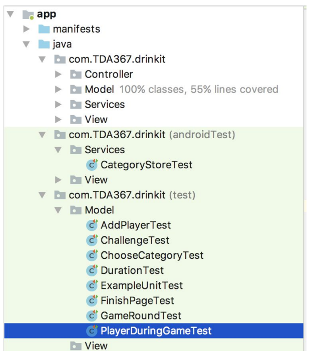
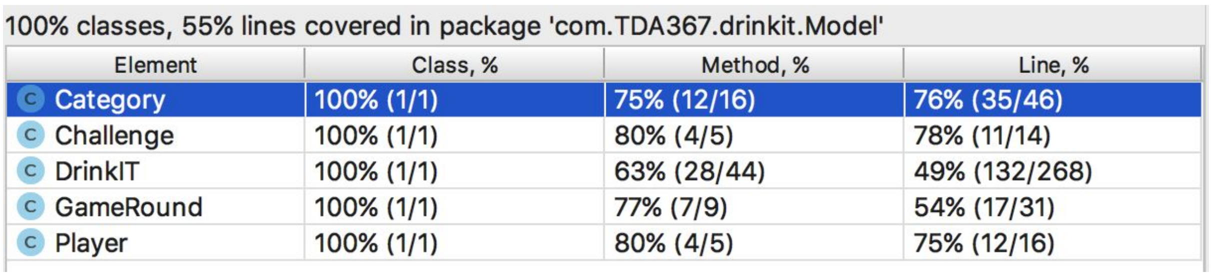

```
▼ app
  ▶ manifests
  ▼ java
    ▼ com.TDA367.drinkit
      ▶ Controller
      ▶ Model  100% classes, 55% lines covered
      ▶ Services
      ▶ View
    ▼ com.TDA367.drinkit (androidTest)
      ▼ Services
          CategoryStoreTest
      ▶ View
    ▼ com.TDA367.drinkit (test)
      ▼ Model
          AddPlayerTest
          ChallengeTest
          ChooseCategoryTest
          DurationTest
          ExampleUnitTest
          FinishPageTest
          GameRoundTest
          PlayerDuringGameTest
        View
```

Figure 6. Where to find the tests



100% classes, 55% lines covered in package 'com.TDA367.drinkit.Model'

Element	Class, %	Method, %	Line, %
Category	100% (1/1)	75% (12/16)	76% (35/46)
Challenge	100% (1/1)	80% (4/5)	78% (11/14)
DrinkIT	100% (1/1)	63% (28/44)	49% (132/268)
GameRound	100% (1/1)	77% (7/9)	54% (17/31)
Player	100% (1/1)	80% (4/5)	75% (12/16)

Figure 7. Coverage of the tests

5.4 Dependency analysis

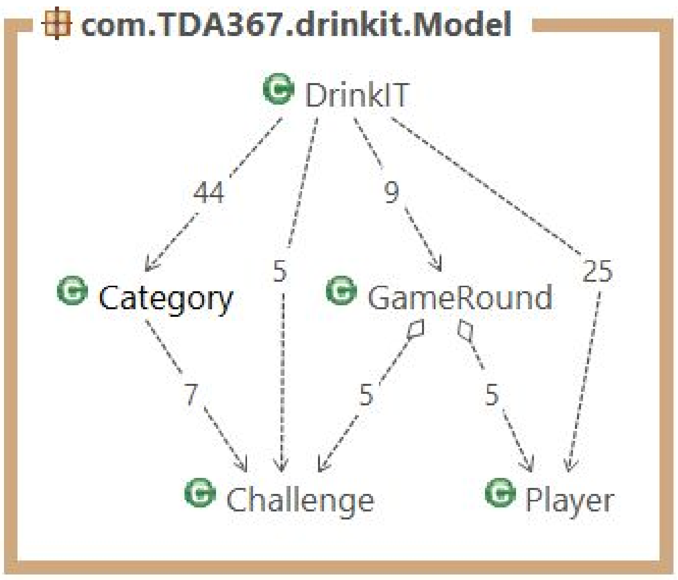


Figure 8. Shows the dependencies in the model package

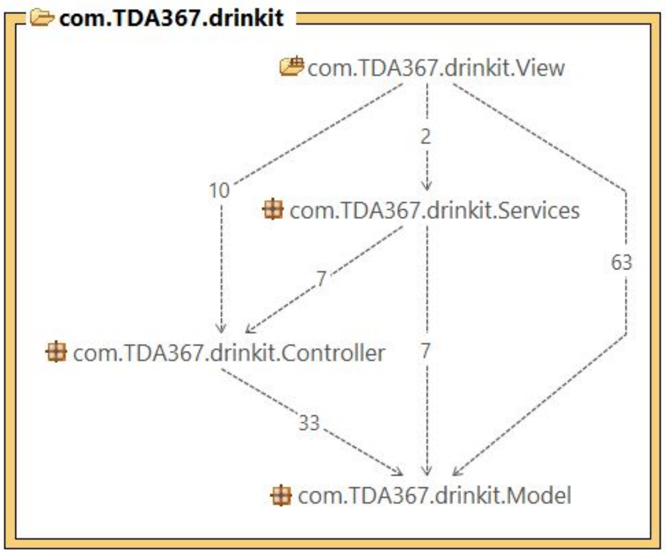


Figure 9. Shows the dependencies between the four packages model, view, controller and services.

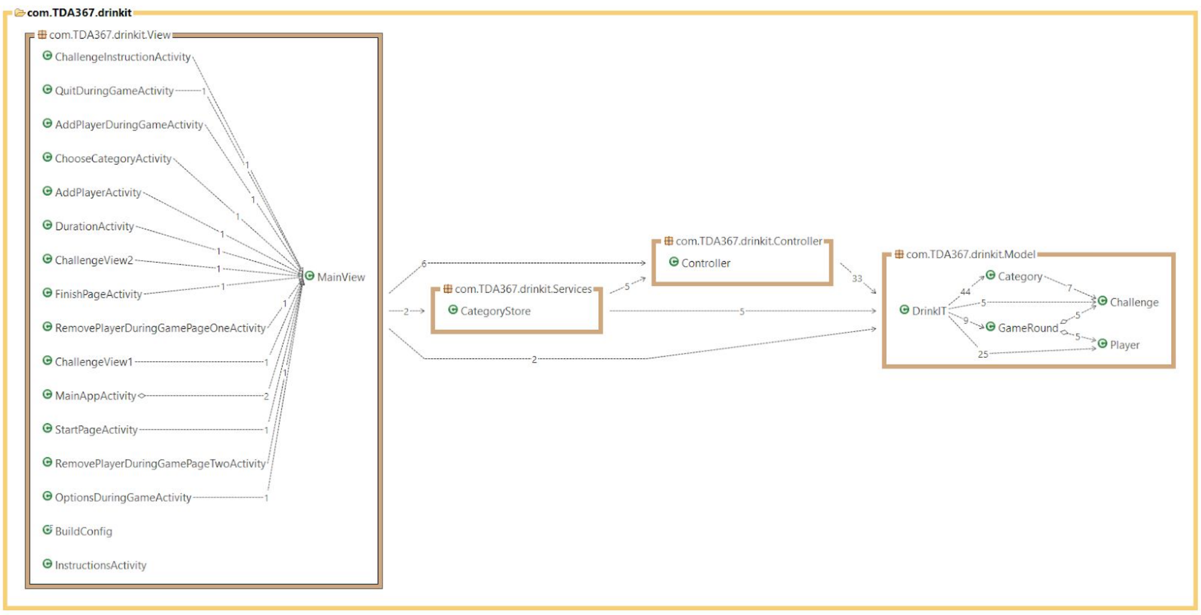


Figure 10. Shows all the dependencies in the project.

6 Persistent data management

Categories are created through files written in JSON format. They are parsed through, and the data in each file is used to generate the categories in the game and all challenges contained within them. Each file contains a name and a description of the category, as well as an array of challenges belonging to the category in question. Each challenge is made up of a challenge text and a point, with some also containing answers to the challenges posed.

When the application is initialized the .json files are parsed through in the class CategoryStore, situated in the package Services. All categories are collected in a list in the model DrinkIT, and its associated challenges are contained in lists in the categories.

This manner of creating categories and challenges lends itself well to further extension of the code. It is easy to add both new categories and new challenges, as the files are parsed through and generated in the same way independent of its size.

7 Potential improvement areas in code management

Like all applications, DrinkIT has both flaws and a potential for further development. There is a lot of potential for future developers concerning this application. Functionality can be added in some fields, for example, regarding the View and some components in the model.

One example of improvement regarding the code concern the class *AddPlayerActivity*. As of now an observer is implemented that checks whether two players have the same name. While the user writes in the name of the players the user gets direct feedback via an error message besides the textfield. The observer however does not work as planned due to the delay of the error message. Only when the user presses "delete" does the error message appear if the same name is already written in another textfield. This is an error which future developers should look over.

Another example concerns the class AddPlayerActivity and ChooseCategoryActivity which consist of a lot of duplicate code. At this moment in time, there is no easy way to add new categories to DrinkIT due to the nine hard coded categories in ChooseCategoryActivity. However this could be arranged differently and made more abstract by implementing methods which gets data from the model instead of just having rows of instantiations.

In FinishPageActivity there is a method, continueTheGame, which is not yet completed. The method should save all the input from the users but as of now it does not.

There is one place in the code where the categories are compared to Strings, which is when loading different views depending on the active category. This is necessary, but it is a good practice to keep such comparisons to the minimum, as it decreases the amount of code that would have to be edited when for example potentially adding a completely new category. It also violates the Open-Closed Principle since Strings are easy to modify.

Lastly, functionality regarding challenges that involves all players should not be directed to a single player. At this time, due to the connection between the list of players and list of categories that is not possible. This could be refactored by future developers.

8 Peer review - Desktop application “Bullet”

The program as a whole and the thoughts behind it feels thought-through and well structured. The group behind the project seems to have a clear picture of the application they want to create, and they feel it is an application that is needed and wanted by many.

The first impression of the code is very overwhelming due to the applications many packages and packages within packages, where some only contain a single class or interface. This however may be due to plans of future expansion where more classes will be added with more functionality. Be that as it may, the number of packages currently appears unnecessary and makes it difficult to navigate the code. The classes themselves seem very structured and thought-through with very clear names.

In many places in the code there are calls to methods that don’t exist, and similarly, there are many methods that have been written but are never used. Our guess it that the group has not yet finished their implementation, and that the application will work and look very differently after this week has passed. This, together with incomplete RAD and SDD make it slightly difficult for an outsider to presently completely understand the code.

The design model gives a clear overview of the application, even though the amount of packages and classes naturally take time to understand. It is however not updated in relation to the code, which makes it difficult to translate between the two. There are many classes, but specifically many packages missing in the design model so far. If the design model had been updated it would probably have been a very good tool in understanding the code in more detail.

As the application is very extensive there is understandably a large amount of dependencies, even some inbetween packages. These are however mainly unidirectional and necessary. Large parts of the model are also wholly independent and reusable. For example the two packages Services and Utils, which both contain functionality which easily can be implemented in other projects.

The documentation of the code is inconsistent. Some classes are void of documentation while some are explained well through javadoc. The package “bullet” is relatively well-documented, but many of the other packages are so far partially lacking in the documentation aspect.

The application follows multiple design patterns and design principles. For example, the class *Bullet* works as a facade for the application. The MVC pattern is followed by dividing the application into six different packages, “bullet”, “controller”, “view”, “util”, “services” and “content”. The model consists of four packages where “Bullet” contains the main logic. The view package consists of multiple packages inside, and is responsible for taking input from the user and also for handling the graphical representation. The Factory pattern is also used to hide internal implementation. An example is BulletPointTreeFactory and SerializerFactory. The Observer pattern is also used multiple times in the view to give dynamic response to the user.

The tests implemented in the project are easy to find in a separate package called “test”. We assume this part of the project is not completed yet, as the coverage of the tests is so far not optimal. The “Bullet” package has a coverage of 28% of the class, and 12% of it’s methods. The other packages have similar test-coverage.

It is not possible to test the GUI completely, as the view does not seem to be finished and/or implemented and connected yet. From the tests and from running the code we can see that most methods are functional, but they are so far not implemented graphically in the view.

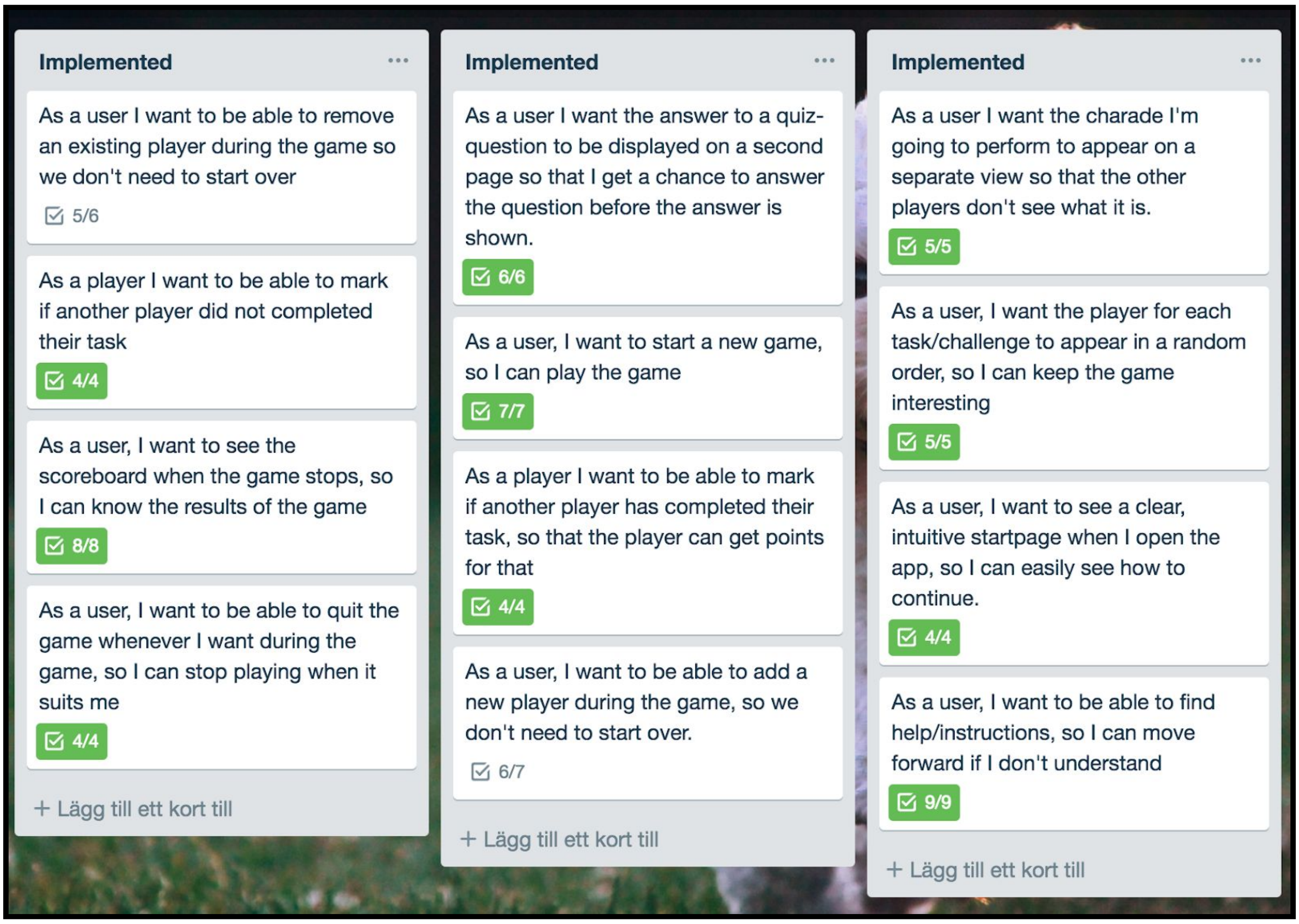
The level of encapsulation in the code is inconsistent and could probably be improved upon. Many public methods could probably instead be set to package-private, and some to private.

In conclusion, Bullet as an application seems to be well thought-through and the developers seem to be both ambitious and capable. It seems the implementation is not completely done yet, and the application will probably seem more complete next week. There are naturally shortcomings, but this is understandable, and overall the application has great potential.

9 References

None.

10 Appendix A: Implemented User Stories



Implemented

As a user I want to be able to remove an existing player during the game so we don't need to start over
☑ 5/6

As a player I want to be able to mark if another player did not completed their task
☑ 4/4

As a user, I want to see the scoreboard when the game stops, so I can know the results of the game
☑ 8/8

As a user, I want to be able to quit the game whenever I want during the game, so I can stop playing when it suits me
☑ 4/4

+ Lägg till ett kort till

Implemented

As a user I want the answer to a quiz-question to be displayed on a second page so that I get a chance to answer the question before the answer is shown.
☑ 6/6

As a user, I want to start a new game, so I can play the game
☑ 7/7

As a player I want to be able to mark if another player has completed their task, so that the player can get points for that
☑ 4/4

As a user, I want to be able to add a new player during the game, so we don't need to start over.
☑ 6/7

+ Lägg till ett kort till

Implemented

As a user I want the charade I'm going to perform to appear on a separate view so that the other players don't see what it is.
☑ 5/5

As a user, I want the player for each task/challenge to appear in a random order, so I can keep the game interesting
☑ 5/5

As a user, I want to see a clear, intuitive startpage when I open the app, so I can easily see how to continue.
☑ 4/4

As a user, I want to be able to find help/instructions, so I can move forward if I don't understand
☑ 9/9

+ Lägg till ett kort till

Figure 11. Implemented User Stories

11 Appendix B: Model Package

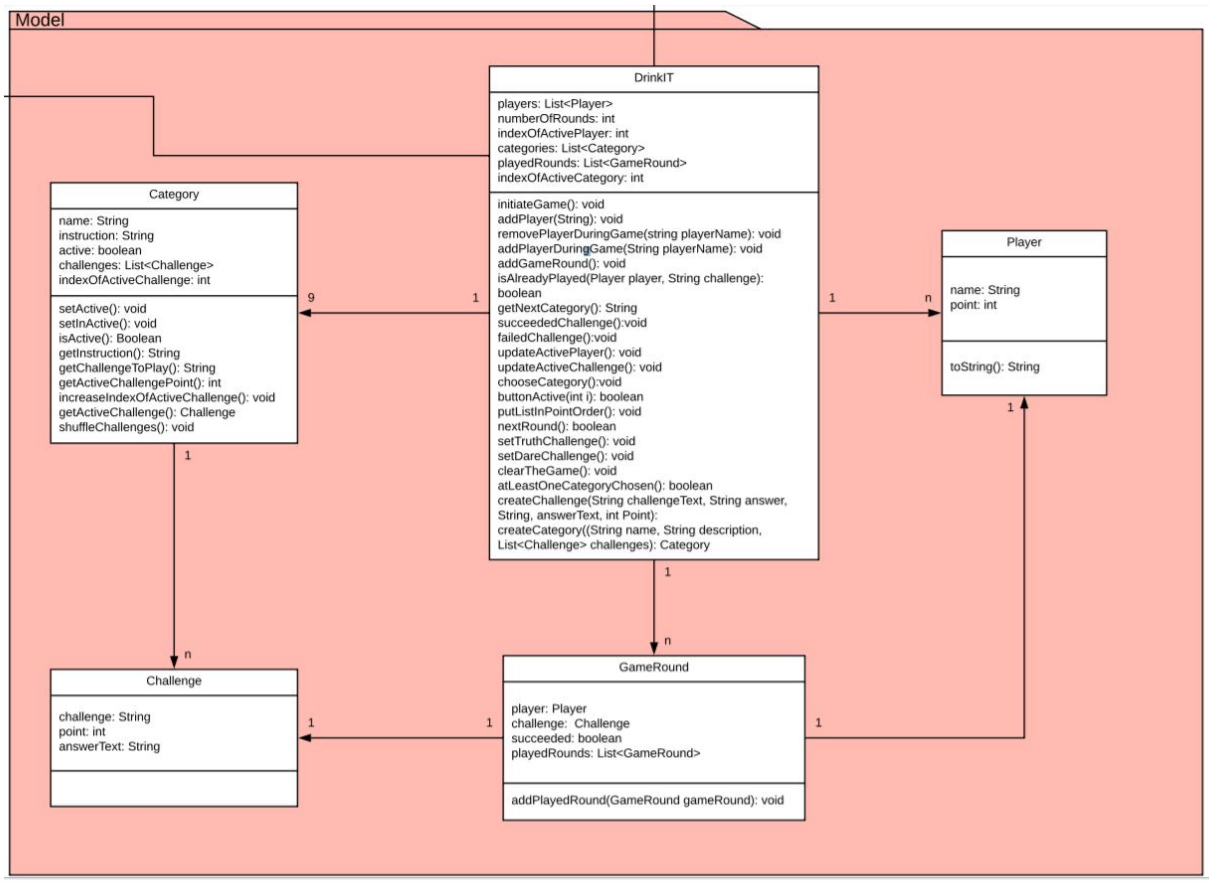

Model
DrinkIT
players: List<Player>
numberOfRounds: int
indexOfActivePlayer: int
categories: List<Category>
playedRounds: List<GameRound>
indexOfActiveCategory: int
initiateGame(): void
addPlayer(String): void
removePlayerDuringGame(string playerName): void
addPlayerDuringGame(String playerName): void
addGameRound(): void
isAlreadyPlayed(Player player, String challenge): boolean
getNextCategory(): String
succeededChallenge():void
failedChallenge():void
updateActivePlayer(): void
updateActiveChallenge(): void
chooseCategory():void
buttonActive(int i): boolean
putListInPointOrder(): void
nextRound(): boolean
setTruthChallenge(): void
setDareChallenge(): void
clearTheGame(): void
atLeastOneCategoryChosen(): boolean
createChallenge(String challengeText, String answer, String, answerText, int Point):
createCategory((String name, String description, List<Challenge> challenges): Category
Category
name: String
instruction: String
active: boolean
challenges: List<Challenge>
indexOfActiveChallenge: int
setActive(): void
setInActive(): void
isActive(): Boolean
getInstruction(): String
getChallengeToPlay(): String
getActiveChallengePoint(): int
increaseIndexOfActiveChallenge(): void
getActiveChallenge(): Challenge
shuffleChallenges(): void
Player
name: String
point: int
toString(): String
Challenge
challenge: String
point: int
answerText: String
GameRound
player: Player
challenge: Challenge
succeeded: boolean
playedRounds: List<GameRound>
addPlayedRound(GameRound gameRound): void
9
1
1
n
1
1
1
n
n
1
1
1

Figure 12. Shows the model package

12 Appendix C: View Package

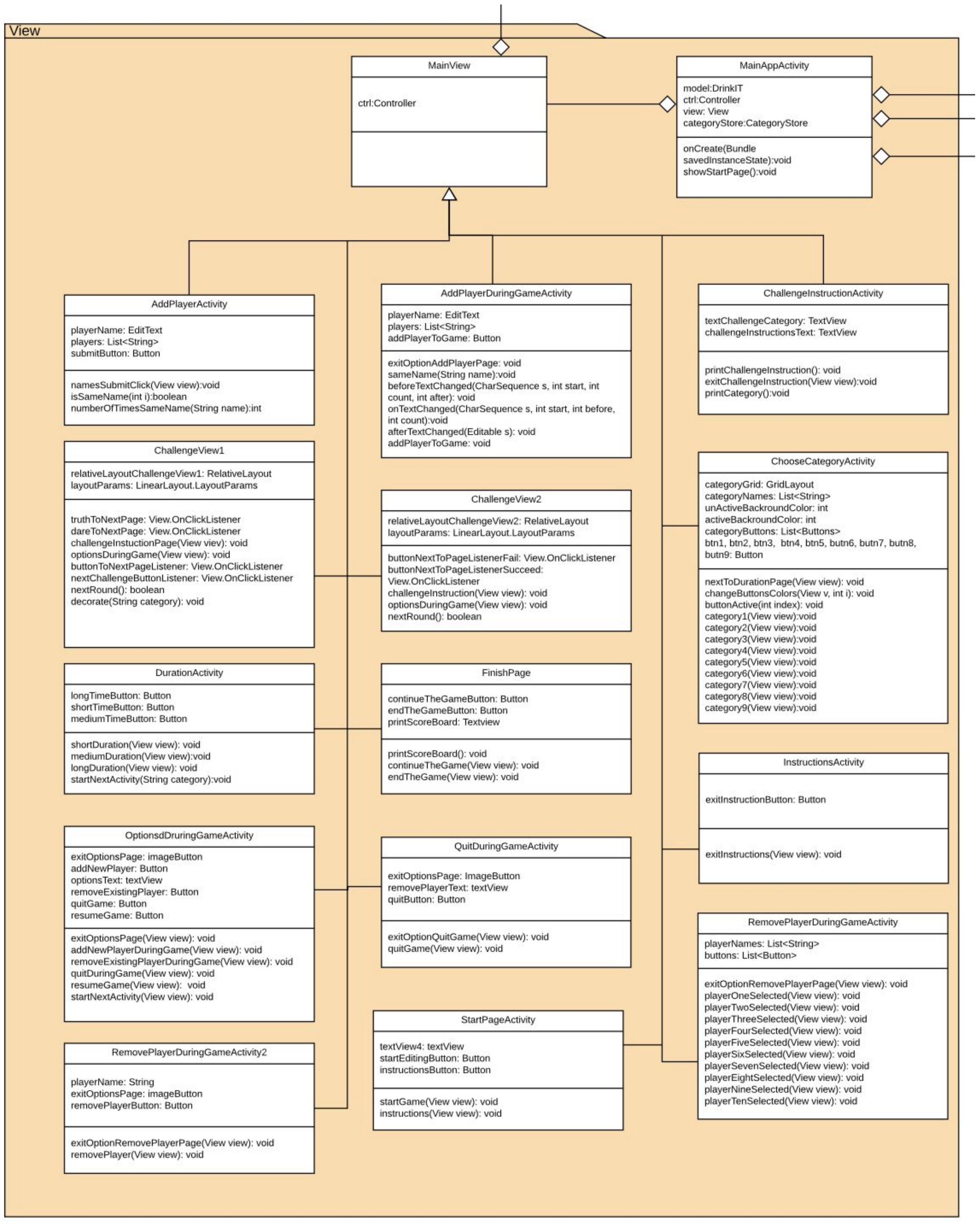

View
MainView
ctrl:Controller
MainAppActivity
model:DrinkIT
ctrl:Controller
view: View
categoryStore:CategoryStore
onCreate(Bundle savedInstanceState):void
showStartPage():void
AddPlayerActivity
playerName: EditText
players: List<String>
submitButton: Button
namesSubmitClick(View view):void
isSameName(int i):boolean
numberOfTimesSameName(String name):int
AddPlayerDuringGameActivity
playerName: EditText
players: List<String>
addPlayerToGame: Button
exitOptionAddPlayerPage: void
sameName(String name):void
beforeTextChanged(CharSequence s, int start, int count, int after): void
onTextChanged(CharSequence s, int start, int before, int count):void
afterTextChanged(Editable s): void
addPlayerToGame: void
ChallengeInstructionActivity
textChallengeCategory: TextView
challengeInstructionsText: TextView
printChallengeInstruction(): void
exitChallengeInstruction(View view):void
printCategory():void
ChallengeView1
relativeLayoutChallengeView1: RelativeLayout
layoutParams: LinearLayout.LayoutParams
truthToNextPage: View.OnClickListener
dareToNextPage: View.OnClickListener
challengeInstuctionPage(View viev): void
optionsDuringGame(View view): void
buttonToNextPageListener: View.OnClickListener
nextChallengeButtonListener: View.OnClickListener
nextRound(): boolean
decorate(String category): void
ChallengeView2
relativeLayoutChallengeView2: RelativeLayout
layoutParams: LinearLayout.LayoutParams
buttonNextToPageListenerFail: View.OnClickListener
buttonNextToPageListenerSucceed: View.OnClickListener
challengeInstruction(View view): void
optionsDuringGame(View view): void
nextRound(): boolean
ChooseCategoryActivity
categoryGrid: GridLayout
categoryNames: List<String>
unActiveBackroundColor: int
activeBackroundColor: int
categoryButtons: List<Buttons>
btn1, btn2, btn3, btn4, btn5, butn6, butn7, butn8, butn9: Button
nextToDurationPage(View view): void
changeButtonsColors(View v, int i): void
buttonActive(int index): void
category1(View view):void
category2(View view):void
category3(View view):void
category4(View view):void
category5(View view):void
category6(View view):void
category7(View view):void
category8(View view):void
category9(View view):void
DurationActivity
longTimeButton: Button
shortTimeButton: Button
mediumTimeButton: Button
shortDuration(View view): void
mediumDuration(View view):void
longDuration(View view): void
startNextActivity(String category):void
FinishPage
continueTheGameButton: Button
endTheGameButton: Button
printScoreBoard: Textview
printScoreBoard(): void
continueTheGame(View view): void
endTheGame(View view): void
InstructionsActivity
exitInstructionButton: Button
exitInstructions(View view): void
OptionsdDruringGameActivity
exitOptionsPage: imageButton
addNewPlayer: Button
optionsText: textView
removeExistingPlayer: Button
quitGame: Button
resumeGame: Button
exitOptionsPage(View view): void
addNewPlayerDuringGame(View view): void
removeExistingPlayerDuringGame(View view): void
quitDuringGame(View view): void
resumeGame(View view): void
startNextActivity(View view): void
QuitDuringGameActivity
exitOptionsPage: ImageButton
removePlayerText: textView
quitButton: Button
exitOptionQuitGame(View view): void
quitGame(View view): void
RemovePlayerDuringGameActivity
playerNames: List<String>
buttons: List<Button>
exitOptionRemovePlayerPage(View view): void
playerOneSelected(View view): void
playerTwoSelected(View view): void
playerThreeSelected(View view): void
playerFourSelected(View view): void
playerFiveSelected(View view): void
playerSixSelected(View view): void
playerSevenSelected(View view): void
playerEightSelected(View view): void
playerNineSelected(View view): void
playerTenSelected(View view): void
StartPageActivity
textView4: textView
startEditingButton: Button
instructionsButton: Button
startGame(View view): void
instructions(View view): void
RemovePlayerDuringGameActivity2
playerName: String
exitOptionsPage: imageButton
removePlayerButton: Button
exitOptionRemovePlayerPage(View view): void
removePlayer(View view): void

Figure 13. Shows the View package.

13 Appendix D: Controller Package

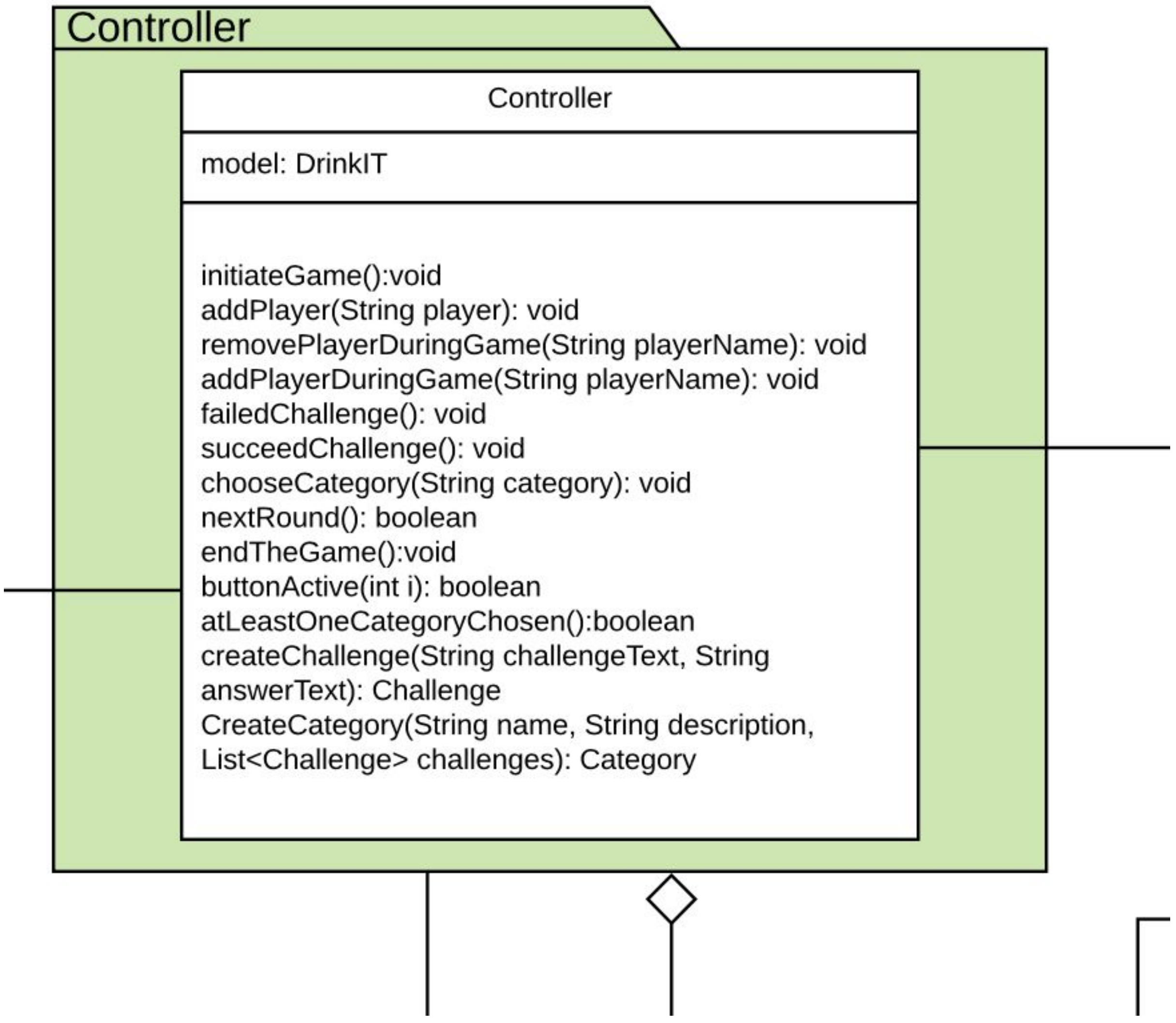

Controller
Controller
model: DrinkIT
initiateGame():void
addPlayer(String player): void
removePlayerDuringGame(String playerName): void
addPlayerDuringGame(String playerName): void
failedChallenge(): void
succeedChallenge(): void
chooseCategory(String category): void
nextRound(): boolean
endTheGame():void
buttonActive(int i): boolean
atLeastOneCategoryChosen():boolean
createChallenge(String challengeText, String answerText): Challenge
CreateCategory(String name, String description, List<Challenge> challenges): Category

Figure 14. Shows the Controller package.